Verifying a Concurrent Garbage Collector with a Rely-Guarantee Methodology

Yannick Zakowski · David Cachera · Delphine Demange · Gustavo Petri · David Pichardie · Suresh Jagannathan · Jan Vitek

Received: date / Accepted: date

Abstract Concurrent garbage collection algorithms are a challenge for program verification. In this paper, we address this problem by proposing a mechanized proof methodology based on the Rely-Guarantee proof technique. We design a compiler intermediate representation with strong type guarantees, dedicated support for abstract concurrent data structures, and high-level iterators on runtime internals. In addition, we define an Rely-Guarantee program logic supporting an incremental proof methodology where annotations and invariants can be progressively enriched. We formalize the intermediate representation, the proof system, and prove the soundness of the methodology in the Coq proof assistant. Equipped with this, we prove a fully concurrent garbage collector where mutators never have to wait for the collector.

1 Introduction

Modern programming languages like ML, Java, and C# rely on garbage collection for the automatic reclamation of memory no longer used by the application. A garbage collector is one of the most subtle parts of modern runtime systems, carefully engineered to minimize performance overheads on application throughput as well as pause times. Some garbage collection algorithms, *on-the-fly* collectors, are able to detect garbage and reclaim it concurrently to the application's threads [5]. These algorithms are notably difficult to implement, test, and prove correct. Unsurprisingly, they also present challenges for mechanized verification. In this paper we focus on algorithms that allow applications threads to operate in parallel with the collector threads, and more specifically we focus on Domani et al.'s lock-free collector [8].

The challenges of formally verifying a garbage collector have been addressed in recent work [10,11,14,15]. Our paper explores a different proof methodology in this space. The

This material is based upon work supported by grants ANR 14-CE28-0004, ERC 695412, NSF CCF-1318227, CCF-1544542, CCF–1618732 and ONR 503353.

Y. Zakowski (✉) · D. Cachera · D. Pichardie · D. Demange, Univ Rennes, Inria, CNRS, IRISA, France
G. Petri, IRIF, Université Paris Diderot
S. Jagannathan, Purdue University, United States
J. Vitek, Northeastern University and Czech Technical University, United States

backbone of our formalization is a new intermediate representation, named RTIR, which we use to implement the garbage collector. Our experience implementing concurrent collectors [31] suggests that choosing the right abstractions is key for reasoning and optimizing these algorithms. The right representation can make the expression and proof of invariants tractable. Moreover, we strive for proofs that can scale to a the verification of a realistic compiler for concurrent, managed languages [4, 17]. RTIR has the following key characteristics: (i) strong type guarantees, (ii) abstract concurrent data structures, (iii) high-level iterators over data structures (iv) support for threads, and (v) support for root management.

Another important characteristic of our approach is the dedicated program logic that accompanies RTIR. While previous approaches [10, 11, 15] attack the proof by means of an abstract state transition system requiring a monolithic invariant, we follow the rely-guarantee [18] (RG) methodology. RG is a major technique for proving the correctness of concurrent programs that provides explicit thread-modular reasoning. Interferences between threads are described using binary relations: *relies* and *guarantees*. Each thread is proved correct under the assumption it is interleaved with other threads fulfilling a *rely* relation. The effect of the thread itself on the shared memory must respect its *guarantee* relation. This guarantee must also be coherent with respect to the relies of other threads. Being able to reason in a modular way is key to a tractable correctness proof, since it avoids explicitly considering all possible interleavings. In contrast, the Owicki-Gries proof technique [28] is generally admitted as less scalable. Crucially, the parallel composition rule in RG abstracts away from the syntax of other threads, while this rule in Owicki-Gries does not. We prove the soundness of our logic, and develop a set of tactics that reduce the proof effort required to discharge the invariants.

Finally, we report on an original *incremental* proof technique. Starting from an implementation, we progressively annotate the program in order to prove stronger and stronger invariants. Each time we enrich the invariant and program annotations, we rely on meta-properties of the proof system, as well as dedicated tactics to discharge proof obligations incrementally, this allows reuse of lemmas proved in prior steps. Our incremental annotation mechanism and its corresponding proof methodology are not specific to garbage collectors. They can be applied more broadly for proofs of concurrent programs relying on RG-like program logics.

Using the Coq proof assistant, we achieved the following: (i) formalizing the syntax and semantics of RTIR and the soundness of an associated RG program logic, (ii) a number of tactics and structural lemmas to facilitate the so-called *stability proofs* required by the RG methodology, (iii) a realistic implementation of Domani et al.'s algorithm [8] in RTIR and (iv) an RG proof ensuring that the collector never frees references accessible by the running threads. Our formal development is available online [42].

2 The RTIR Intermediate Representation

This section gives an overview of our *intermediate representation*, RTIR, which can be thought of as playing a similar role to Java's bytecode representation as a target for compilation from source code. In a complete system, RTIR would be used to compile the client code written in the source language, as well as to compile the *implementation* of the source language runtime systems. Thus both application code and the implementation of the garbage collector would be compiled to RTIR. The reason why a dedicated language is needed is that representations like Java bytecode do not expose operations to access all fields of an object or operations to visit the roots of a thread. For our purposes, we focus on the subset of op-

$X, Y \in \texttt{gvar} \quad x, y \in \texttt{lvar} \quad t, m, C \in \texttt{tid} \quad f \in \texttt{fid} \quad rn \in \texttt{list fid}$

$$
\begin{array}{rcllllll}
\texttt{expr} \ni e & := & const \in \mathbb{Z} & | & \texttt{null} & | & x \quad | \quad \texttt{!e} & | \quad e_1 \ bop \ e_2 \\
\texttt{cmd} \ni c & := & \texttt{skip} & | & \texttt{assume}\ e & | & x = e & \\
 & | & c_1 \ ; \ c_2 & | & c_1 \oplus c_2 & | & \texttt{loop}(c) & | \quad \texttt{atomic}\ c \\
 & | & x = \texttt{alloc}(rn) & | & \texttt{free}(x) & | & x = \texttt{isFree}(y) & \\
 & | & x = Y & | & X = e & | & x = y.f & | \quad x.f = e \\
 & | & x = y.\texttt{isEmpty}() & | & x = y.\texttt{top}() & | & X = y.\texttt{copy}() & \\
 & | & x.\texttt{push}(y) & | & x.\texttt{pop}() & & & \\
 & | & \texttt{foreach}\ (x\ \textit{in}\ \texttt{l})\ \texttt{do}\ c\ \texttt{od} & & & | & \texttt{foreachField}\ (f\ \texttt{of}\ x)\ \texttt{do}\ c\ \texttt{od} & \\
 & | & \texttt{foreachRef}\ r \in \mathcal{R}\ \texttt{when}\ \texttt{P}(r)\ \texttt{do}\ c\ \texttt{od} & & & & & \\
 & | & \texttt{foreachObject}\ x\ \texttt{do}\ c\ \texttt{od} & & & | & \texttt{foreachRoot}\ (x\ \texttt{of}\ t)\ \texttt{do}\ c\ \texttt{od} &
\end{array}
$$

Figure 1: Simplified Syntax of RTIR.

erations that are needed to implement a garbage collector. In a complete system, additional instructions would be provided.

In the remainder, we will write *client code* when referring to application logic compiled to RTIR. This is also what is referred to as the mutator in the garbage collection literature. On the other hand, *injected code* will refer to the code added by the compiler to support garbage collection and also to the implementation of the collector algorithm. In most modern collectors, the collection work is partly offloaded to the mutator threads.

2.1 Syntax

Figure 1 shows the syntax of RTIR. It has two kinds of variables: *global* (shared) variables, in `gvar`, accessed by all threads, and *local* variable, in `lvar`, used for thread-local computations. Side-effect-free expressions ($e \in$ `expr`) are built from integers literals, the special constant `null`, local variables, and the usual binary, arithmetic and comparison operators (*bop*). Booleans are encoded as integers. Commands include instructions, such as `skip`, `assume` e, local variable update $x = e$, and combinators: sequencing, non-deterministic choice ($c_1 \oplus c_2$), and non-deterministic loops. We use macros to define higher level constructs. For instance, the usual conditional (`if` e `then` c_1 `else` c_2) can be defined as (`assume` $e; c_1$) $\oplus$ (`assume` $!e; c_2$), where we write $!e$ for the boolean negation of e. While and repeat-until loops can be encoded similarly: `while` b `do` c `od` $\triangleq$ $loop(assume\ b; c); assume\ !b$. RTIR also provides atomic blocks (`atomic` c). In the implementation of the garbage collector, atomic blocks are only used to add ghost-code, code that does not take part in the computation, and to model linearizable data structures. These atomic constructs can be refined into low-level, fine-grained implementations using techniques from our previous work [17, 43].

RTIR does not allow pointers to be forged. No reference constant literal is available other than `null`. New references are allocated by `alloc`(rn), and de-allocated by `free`. The instruction `isFree` allows to test whether a reference is allocated. Instructions related to shared-memory accesses are fine-grained, i.e. they perform exactly one global operation (read or write). This allows considering each possible interleaving of memory operations arising from different threads in the proofs, while keeping the semantics reasonably simple. RTIR also provides abstract concurrent queues to implement the *mark buffers* needed by Domani et al. [8]. These are accessible through operations $x = y$.`top`(), x.`pop`(), x.`push`(y),

$x = y$.`isEmpty`(). While one could implement buffers in RTIR, we argue it is better to reason about them at a higher level, and hence to assume they act atomically. Implementing these data structures in a linearizable [16] fashion is an orthogonal problem that we addressed in [43]. Buffers also provide an operation $X = y$.`copy`() to perform a deep copy. This operation is only used in ghost-code, and thus never executed at run-time.

A key ingredient of RTIR is its support for *iterators*, making it possible to easily express many bookkeeping tasks of the collector. Two primitive iterators allow for the definition of higher level ones. First, iterator `foreach` (*x in* `l`) `do` *c* `od`, where variable x can be free in command c, iterates c through all elements of the static list `l`. Binder x is represented here in the syntax for sake of clarity, but is actually a metavariable bound at the Coq level: command c is actually a function of type `A` $\rightarrow$ `cmd` where `A` is the type of the elements over which we iterate. We will come back to this technicality when describing the semantics. Second, iterator `foreachRef` $r \in \mathcal{R}$ `when` `P`(r) `do` c `od`, where c is a function of type `ref` $\rightarrow$ `cmd`, iterates over a set of references $\mathcal{R}$. This set is computed at runtime by a predicate `P` which filters the references over which iteration occurs. Iterators simplify the definition of more sophisticated bookkeeping tasks. Iteration over a static set of values includes iteration over thread identifiers and fields of a given object. Iteration over a dynamically computed set is used to visit the local variables of a thread (i.e. *roots*) and the objects in the heap.

2.2 Operational semantics

The soundness of our logic, and hence the correctness theorem of our garbage collector, is phrased in terms of the operational semantics of RTIR. As RTIR is concurrent, a small-step semantics is needed to express the interleaving of threads, and a big-step semantics will be used to give meaning to atomic blocks.

2.2.1 Typing information

The semantics is enriched with typing information. Basic types in `typ` include `TNum` for numeric constants, `TRef` for references to objects, and `TRefSet` for non-null references to abstract mark-buffers. Local variables, global variables, and field identifiers are declared to have exactly one of these types. Additionally, a boolean flag marks local variables to indicate whether they are reserved for use by the injected code, or belong to the client code. Variable are thus encoded in Coq as records with three fields. Types are retrieved through the corresponding accessor, respectively `lvar_typ`, `gvar_typ` and `fid_typ`, while the accessor `user` retrieves the boolean flag associated to a local variable.

RTIR manipulates two kinds of values: numeric values in the Coq type `Z` and references in `ref`. Types are mapped to values with the function `value` of type `typ` $\rightarrow$ `Type`.

`typ` $\triangleq$ { `TNum`, `TRef`, `TRefSet` }
`lvar` $\triangleq$ `varId` $\times$ `typ` $\times$ `bool`
`gvar` $\triangleq$ `varId` $\times$ `typ`
`fid` $\triangleq$ `fieldId` $\times$ `typ`

```
Definition value (t:typ) : Type :=
  match t with
  | TNum ⇒ Z
  | TRef | TRefSet ⇒ ref
  end.
```

2.2.2 Semantic domains

Local (resp. global) environments map local (resp. global) variables to values of their declared type. Environments are hence dependent functions of type:

```
Definition lenv := ∀ x:lvar, value (lvar_typ x).
Definition genv := ∀ X:gvar, value (gvar_typ X).
```

A thread-local state is defined by a local environment and a command to execute. A global state includes a global environment `ge` and a heap `hp` – a partial map from references to `objects`. We consider two distinct kinds of objects: regular objects, mapping fields to values, and abstract mark-buffers. Global states also include two components essential to the implementation of a garbage collector: `roots` and a `freelist`. The freelist is a shared data structure, while roots are considered to be thread-local – mutators are responsible for handling their own roots, under the form of reference counters. Here, we model roots as part of the global state only to ease proof annotations – our final theorem is indeed an invariant of the program's global state.

```
Definition thread_state := (cmd * lenv).
Record gstate := {  ge: genv;                  freelist: ref → bool;
                    hp: ref → option object;   roots: tid → ref → nat }.
```

Finally, an execution state includes the states of all threads and a global state.

```
Definition state := ((tid → option thread_state) * gstate).
```

2.2.3 Semantics of atomic instructions

RTIR has two operational semantics: a *big-step* semantics and a *small-step* interleaving semantics. The big-step semantics has two essential uses. First, it defines the semantic validity of Hoare-like tuples for basic instructions in our proof system (see section 3). Secondly, it is used by the small-step semantics to define the meaning of commands in atomic blocks. The small-step semantics on the other hand interleaves the execution of the threads, and is used to prove our final soundness results. Both semantics share the reduction rules for elementary, inherently atomic, instructions. Figure 2 gives these rules for the reduction relation $(c \mid s) \rightsquigarrow_t s'$ of a thread t: given execution state s, command c fully executes in a step and produces execution state s'.

Let us introduce some notation. We use meta-variables $\rho \in$ `lenv` for local maps and $\sigma = (ge, hp, fl, rt) \in$ `gstate` for the components of a global state. The map resulting from the update of m with a new binding of v to x is written $m[x \leftarrow v]$. Updates of the `roots` mapping are handled specially. Operation `roots`$[rt, t, x, v_o, v_n]$ checks the type of variable x: if it is `TRef`, then it decrements, for thread t, the counter of reference v_o and increments the one of reference v_n; otherwise it leaves the roots component of the local state unchanged. To keep rules compact, we also use set-theoretic notations to handle the freelist. Buffers are manipulated as lists of references, where the empty list is written `nil` and :: denotes the `cons` operator. When a map h is partial, we write $h(x) \doteq v$ as a shorthand to $h(x) = \mathit{Some}\ v$. We write $r.(f)$ to access field f of record r, and $r\{f[x \leftarrow v]\}$ to denote record r in which field f is updated by value v. The language is dynamically typed, but in order to avoid a systematic mention of typing checks in the rules, we enforce a strict use of meta-variables. References, buffers and integers are respectively denoted by metavariables `r`, `s` and `b`, while metavariable `v` may refer to any value. As an example, an assumption like $\rho(x) = r$ or $\rho[x \leftarrow r]$ implicitly carries the side-condition that `lvar_typ`$(x) =$ `TRef`. Finally, we assume a straightforward semantics $[\![\cdot]\!]$ for expressions.

The semantics of atomic instructions is standard but for root management. Local computations $x = e$ have two different rules, depending on whether x is a client variable or not: in the former case we need to additionally update the roots. Note that the semantics hence

$$\frac{}{((\rho,\sigma) \mid \texttt{skip}) \rightsquigarrow_t (\rho,\sigma)} \qquad \frac{[\![b]\!]\rho = true}{((\rho,\sigma) \mid \texttt{assume}\ b) \rightsquigarrow_t (\rho,\sigma)} \qquad \frac{x.(user) = false \quad [\![e]\!]\rho = v_e}{((\rho,\sigma) \mid x = e) \rightsquigarrow_t (\rho[x \leftarrow v_e],\sigma)}$$

$$\frac{x.(user) = true \quad [\![e]\!]\rho = v_e \quad \rho(x) = v_x}{((\rho,(ge,hp,fl,rt)) \mid x = e) \rightsquigarrow_t (\rho[x \leftarrow v_e],(ge,hp,fl,\texttt{roots}[rt,t,x,v_x,v_e]))}$$

$$\frac{X.(gvar\_typ) \neq \texttt{TRefSet} \quad [\![e]\!]\rho = v}{((\rho,(ge,hp,fl,rt)) \mid X = e) \rightsquigarrow_t (\rho,(ge[X \leftarrow v],hp,fl,rt))} \qquad \frac{x.(user) = false \quad ge(Y) = v}{((\rho,\sigma) \mid x = Y) \rightsquigarrow_t (\rho[x \leftarrow v],\sigma)}$$

$$\frac{x.(user) = true \quad ge(Y) = v_n \quad \rho(x) = v_o}{((\rho,(ge,hp,fl,rt)) \mid x = Y) \rightsquigarrow_t (\rho[x \leftarrow v_n],(ge,hp,fl,\texttt{roots}[rt,t,x,v_o,v_n]))}$$

$$\frac{[\![e]\!]\rho = v \quad \rho(x) = r_x \quad hp(r_x) \doteq ob}{((\rho,(ge,hp,fl,rt)) \mid x.f = e) \rightsquigarrow_t (\rho,(ge,hp[r_x \leftarrow ob[f \leftarrow v]],fl,rt))}$$

$$\frac{\rho(y) = r \quad hp(r) \doteq ob \quad ob(f) \doteq v_n \quad \rho(x) = v_o}{((\rho,(ge,hp,fl,rt)) \mid x = y.f) \rightsquigarrow_t (\rho[x \leftarrow v_n],(ge,hp,fl,\texttt{roots}[rt,t,x,v_o,v_n]))}$$

$$\frac{r \in fl \quad hp' = hp[r \leftarrow init_{obj}(rn)] \quad fl' = fl \setminus \{r\}}{((\rho,(ge,hp,fl,rt)) \mid x = \texttt{alloc}(rn)) \rightsquigarrow_t (\rho[x \leftarrow r],(ge,hp',fl',rt))}$$

$$\frac{}{((\rho,(ge,hp,fl,rt)) \mid \texttt{free}(x)) \rightsquigarrow_t (\rho,(ge,hp[r \leftarrow None],fl \cup \{r\},rt))}$$

$$\frac{\rho(y) = r \quad b = (r \in fl)}{((\rho,\sigma) \mid x = \texttt{isFree}(y)) \rightsquigarrow_t (\rho[x \leftarrow b],\sigma)} \qquad \frac{hp(r) \doteq ob}{((\rho,\sigma) \mid x \leftarrow_{ref} r) \rightsquigarrow_t (\rho[x \leftarrow r],\sigma)}$$

$$\frac{\rho(x) = r_x \quad \rho(y) = r_y \quad hp(r_x) \doteq s}{((\rho,(ge,hp,fl,rt)) \mid x.\texttt{push}(y)) \rightsquigarrow_t (\rho,(ge,hp[r_x \leftarrow r_y :: s],fl,rt))}$$

$$\frac{\rho(x) = r_x \quad hp(r_x) \doteq (r :: s)}{((\rho,(ge,hp,fl,rt)) \mid x.\texttt{pop}()) \rightsquigarrow_t (\rho,(ge,hp[r_x \leftarrow s],fl,rt))}$$

$$\frac{\rho(y) = r_y \quad hp(r_y) \doteq (r :: s)}{((\rho,\sigma) \mid x = y.\texttt{top}()) \rightsquigarrow_t (\rho[x \leftarrow r],\sigma)} \qquad \frac{\rho(y) = r_y \quad hp(r_y) \doteq s \quad b = (s = nil)}{((\rho,\sigma) \mid x = y.\texttt{isEmpty}()) \rightsquigarrow_t (\rho[x \leftarrow b],\sigma)}$$

$$\frac{ge(X) = r_x \quad \rho(y) = r_y \quad hp(r_y) \doteq s}{((\rho,(ge,hp,fl,rt)) \mid X = y.\texttt{copy}()) \rightsquigarrow_t (\rho,(ge,hp[r_x \leftarrow s],fl,rt))}$$

Figure 2: Semantics of atomic instructions.

depends on the thread's identifier. Loads and stores to the heap are similar to their global counterparts, except for the fact that the semantics is blocking if we try to access references outside of the heap domain, or fields outside of the domain of the object.

Allocation $x = \texttt{alloc}(rn)$ non-deterministically picks a reference r from the freelist. A fresh object $init_{obj}(rn)$ is bound to r in the heap: the domain of the object is the set of fields provided in rn, and each field is bound to either 0 or the *Null* reference depending on its type. Finally, r is removed from the freelist. Freeing a reference r removes it from the domain of the heap and returns it to the freelist.

Next we consider abstract operations over buffers. The `push` and `pop` operations update a buffer, `pop` blocks if the buffer is empty. The `top` and `isEmpty` operations are straightforward. The `copy` instruction performs a deep-copy of the buffer and stores it in the heap. This op-

$$\frac{(s_1 \mid c) \rightsquigarrow_t s_2}{(s_1 \mid c) \Downarrow_t s_2} \qquad \frac{(s_1 \mid c_1) \Downarrow_t s_2 \quad (s_2 \mid c_2) \Downarrow_t s_3}{(s_1 \mid c_1; c_2) \Downarrow_t s_3} \qquad \frac{(s_1 \mid \texttt{skip} \oplus (c; \texttt{loop}(c))) \Downarrow_t s_2}{(s_1 \mid \texttt{loop}(c)) \Downarrow_t s_2}$$

$$\frac{(s_1 \mid c_1) \Downarrow_t s_2}{(s_1 \mid c_1 \oplus c_2) \Downarrow_t s_2} \qquad \frac{(s_1 \mid c_2) \Downarrow_t s_2}{(s_1 \mid c_1 \oplus c_2) \Downarrow_t s_2} \qquad \frac{(s_1 \mid c) \Downarrow_t s_2}{(s_1 \mid \texttt{atomic}\ c) \Downarrow_t s_2}$$

$$\frac{}{(s \mid \texttt{foreach}\ (x\ \textit{in}\ \texttt{nil})\ \texttt{do}\ c\ \texttt{od}) \Downarrow_t s} \qquad \frac{(s_1 \mid c(a); \texttt{foreach}\ (x\ \textit{in}\ S)\ \texttt{do}\ c\ \texttt{od}) \Downarrow_t s_2}{(s_1 \mid \texttt{foreach}\ (x\ \textit{in}\ (a :: S))\ \texttt{do}\ c\ \texttt{od}) \Downarrow_t s_2}$$

$$\frac{\texttt{ref\_fields}(ob) = S \quad (s_1 \mid \texttt{foreach}\ (f\ \textit{in}\ S)\ \texttt{do}\ c\ \texttt{od})) \Downarrow_t s_2}{(s_1 \mid \texttt{foreachField}\ (f\ \texttt{of}\ ob)\ \texttt{do}\ c\ \texttt{od}) \Downarrow_t s_2}$$

$$\frac{}{(s \mid \texttt{foreachRef}\ x \in \texttt{nil}\ \texttt{when}\ P(x)\ \texttt{do}\ c\ \texttt{od}) \Downarrow_t s}$$

$$\frac{P(a) = \textit{false} \quad (s_1 \mid \texttt{foreachRef}\ x \in S\ \texttt{when}\ P(x)\ \texttt{do}\ c\ \texttt{od}) \Downarrow_t s_2}{(s_1 \mid \texttt{foreachRef}\ x \in a :: S\ \texttt{when}\ P(x)\ \texttt{do}\ c\ \texttt{od}) \Downarrow_t s_2}$$

$$\frac{P(a) = \textit{true} \quad (s_1 \mid x \leftarrow_{\textit{ref}} a; c(a); \texttt{foreachRef}\ x \in S\ \texttt{when}\ P(x)\ \texttt{do}\ c\ \texttt{od}) \Downarrow_t s_2}{(s_1 \mid \texttt{foreachRef}\ x \in (a :: S)\ \texttt{when}\ P(x)\ \texttt{do}\ c\ \texttt{od}) \Downarrow_t s_2}$$

$$\frac{\textit{AllObjects}(O) \quad (s_1 \mid \texttt{foreachRef}\ o \in O\ \texttt{when}\ \texttt{in\_heap}(o)\ \texttt{do}\ c\ \texttt{od}) \Downarrow_t s_2}{(s_1 \mid \texttt{foreachObject}\ o\ \texttt{do}\ c\ \texttt{od}) \Downarrow_t s_2} \qquad \frac{\textit{AllRoots}_t(\mathcal{R}) \quad (s_1 \mid \texttt{foreachRef}\ r \in \mathcal{R}\ \texttt{when}\ \texttt{True}(r)\ \texttt{do}\ c\ \texttt{od}) \Downarrow_t s_2}{(s_1 \mid \texttt{foreachRoot}\ (r\ \texttt{of}\ t)\ \texttt{do}\ c\ \texttt{od}) \Downarrow_t s_2}$$

Figure 3: Big-step semantics.

eration is the only that is not atomic: it only occurs in the ghost-code used for proofs, most specifically in `trace` described in section 4.3.

Last, the $\leftarrow_{ref}$ instruction can never appear in the client or the injected code, as no reference literals are available in the language. Rather, we use this pseudo-instruction to define the semantics of the `foreachRef` iterator. The $\leftarrow_{ref}$ instruction is used to bind a reference to a variable, and the reference constant appearing in that instruction is built by semantically unfolding the iterator.

2.2.4 Big-step semantics

Figure 3 gives the sequential big-step semantics of commands, whose judgments are of the form $(s_1 \mid c) \Downarrow_t s_2$. This big-step semantics is defined on top of the semantics of elementary commands. The control flow is traditional, the non-deterministic loop being recursively expressed in terms of a non-deterministic choice with `skip`. The semantics of an atomic block is expressed as the semantics of its contents.

Iterators are interesting. First, `foreach` (*x in S*) `do` *c* `od` iterates a command *c* through a list `S` of *A*s. The semantics are as expected for traditional loops. However, we resort to *higher-order abstract syntax* [30] to define iterators: *x* is not a program variable, but rather bound at the Coq level by parameterizing *c* by an element of type *A*, thus *c* has type `A → cmd`. This has several benefits. It avoids over-constraining the type *A*: using a regular variable would restrict *A* to RTIR values. Our approach allows iterating over introspective values without introducing their types into the language. We use this construct to iterate over thread identifiers, as well as to define a dedicated iterator over the fields of an object,

$$\frac{(s_1 \mid c) \rightsquigarrow_t s_2}{(s_1 \mid c) \rightarrow_t (s_2 \mid \texttt{skip})} \qquad \frac{(s_1 \mid c_1) \rightarrow_t (s_2 \mid c_1')}{(s_1 \mid c_1; c_2) \rightarrow_t (s_2 \mid c_1'; c_2)} \qquad \frac{}{(s \mid \texttt{skip}; c_2) \rightarrow_t (s \mid c_2)}$$

$$\frac{}{(s \mid \texttt{loop}(c)) \rightarrow_t (s \mid \texttt{skip} \oplus (c; \texttt{loop}(c)))} \qquad \frac{}{(s \mid c_1 \oplus c_2) \rightarrow_t (s \mid c_1)} \qquad \frac{}{(s \mid c_1 \oplus c_2) \rightarrow_t (s \mid c_2)}$$

$$\frac{(s_1 \mid c) \Downarrow_t s_2}{(s_1 \mid \texttt{atomic}\ c) \rightarrow_t (s_2 \mid \texttt{skip})} \qquad \frac{}{(s \mid \texttt{foreach}\ (x\ \textit{in}\ \texttt{nil})\ \texttt{do}\ c\ \texttt{od}) \rightarrow_t (s \mid \texttt{skip})}$$

$$\frac{}{(s \mid \texttt{foreach}\ (x\ \textit{in}\ (a :: S))\ \texttt{do}\ c\ \texttt{od}) \rightarrow_t (s \mid c(a); \texttt{foreach}\ (x\ \textit{in}\ S)\ \texttt{do}\ c\ \texttt{od})}$$

$$\frac{\texttt{ref\_fields}(ob) = S}{(s \mid \texttt{foreachField}\ (f\ \texttt{of}\ ob)\ \texttt{do}\ c\ \texttt{od}) \rightarrow_t (s \mid \texttt{foreach}\ (f\ \textit{in}\ S)\ \texttt{do}\ c\ \texttt{od})}$$

$$\frac{}{(s \mid \texttt{foreachRef}\ x \in \texttt{nil}\ \texttt{when}\ P(x)\ \texttt{do}\ c\ \texttt{od}) \rightarrow_t (s \mid \texttt{skip})}$$

$$\frac{P(a) = \textit{false}}{(s \mid \texttt{foreachRef}\ x \in a :: S\ \texttt{when}\ P(x)\ \texttt{do}\ c\ \texttt{od}) \rightarrow_t (s \mid \texttt{foreachRef}\ x \in S\ \texttt{when}\ P(x)\ \texttt{do}\ c\ \texttt{od})}$$

$$\frac{P(a) = \textit{true}}{(s \mid \texttt{foreachRef}\ x \in (a :: S)\ \texttt{when}\ P(x)\ \texttt{do}\ c\ \texttt{od}) \rightarrow_t (s \mid x \leftarrow_{\textit{ref}} a; c(a); \texttt{foreachRef}\ x \in S\ \texttt{when}\ P(x)\ \texttt{do}\ c\ \texttt{od})}$$

$$\frac{\textit{AllObjects}(O)}{(s \mid \texttt{foreachObject}\ o\ \texttt{do}\ c\ \texttt{od}) \rightarrow_t (s \mid \texttt{foreachRef}\ o \in O\ \texttt{when}\ \texttt{in\_heap}(o)\ \texttt{do}\ c\ \texttt{od})}$$

$$\frac{\textit{AllRoots}_t(\mathcal{R})}{(s \mid \texttt{foreachRoot}\ (r\ \texttt{of}\ t)\ \texttt{do}\ c\ \texttt{od}) \rightarrow_t (s \mid \texttt{foreachRef}\ r \in \mathcal{R}\ \texttt{when}\ \texttt{True}(r)\ \texttt{do}\ c\ \texttt{od})}$$

Figure 4: Small-step semantics.

`foreachField` (f `of` ob) `do` c `od`. The iteration list for objects is computed at runtime through predicate `ref_fields`(ob) which returns the list of fields of object ob holding a value of type `TRef`. The iterator then executes command `(c f)` for all fields `f` of this list.

More sophisticated examples of bookkeeping tasks, such as visiting the fields of an object or the roots of a thread, use dynamically computed lists of references. For these, `foreachRef` $x \in S$ `when` $P(x)$ `do` c `od` iterates over references S with a predicate P to filter unwanted references. At each iteration, the current reference is bound to a local variable through the dedicated instruction $\leftarrow_{ref}$. The body of the iterator, c, can refer to the current reference through this variable. Nonetheless, c is parameterized by the reference to ease the definition of proof annotations. The `foreachRef` iterator is used to define the `foreachObject` iterator, where predicate *AllObjects* characterizes the list of all references to objects initially allocated in the heap; at each iteration step, predicate `in_heap` filters objects which would have been freed. Finally, `foreachRoot` (r `of` t) `do` c `od` iterates over all roots of a thread t, which is determined at runtime by predicate $\textit{AllRoots}_t$.

2.2.5 Small-step semantics

The small-step semantics of RrIR is denoted by $\rightarrow_t$, where t is the stepping thread's id. The rules, given in Figure 4, mirror the big-step semantics in a small-step style. The reduction

step for an atomic block makes use of the big-step semantics to prohibit any interleaving with other threads inside of the block. Finally, the interleaving semantics $\hookrightarrow$ can be defined as a binary relation over execution states:

$$\frac{tss(t) = (c_1, \rho_1) \qquad ((\rho_1, \sigma_1) \mid c_1) \rightarrow_t ((\rho_2, \sigma_2) \mid c_2)}{(tss \mid \sigma_1) \hookrightarrow (tss[t \leftarrow (c_2, \rho_2)] \mid \sigma_2)}$$

2.2.6 Well-typedness invariants

To ease proofs, we embed several well-typedness constraints in the small-step rules. These constraints are of three forms:

Basic typing: Variables in the local or global environment contain a value that is of their declared type. This semantic property could be proved via a simple type system.

Singleton property: Each abstract mark-buffer is accessible from a unique global variable, indexed by a thread identifier. This mechanism enforces separation of mark-buffers. To statically enforce this property, we could set up a dedicated static analysis to check that (a) during buffers allocation, fresh buffer addresses are assigned to dedicated global variables, and (b) these global variables are never modified.

No dangling references: Any reference of type `TRef` is either null, in the domain of the heap, or in the freelist. This can be shown to be a semantic invariant without extra static constraints.

3 RTIR Proof System

On top of RTIR, we design a variant of the rely-guarantee (RG) program logic. In a nutshell, RG [18] extends Hoare-logic to handle concurrency in a thread modular fashion. In addition to the standard Hoare-tuples, side conditions ensure that program annotations take into account the possible interferences of other threads. When thinking about a particular thread's code, we shall refer to the actions of the other concurrent threads as its *context*. This context is formally encoded as a *rely* relation stating its possible execution steps. Thus, each annotation in the code of a thread must be proved to be *stable w.r.t.* its rely condition, meaning that its validity is not affected by possible state changes induced by any number of rely steps.

3.1 Design choices for proof rules

In our approach, we first annotate a program and then prove the annotated program using syntax-directed proof rules. To this end, the syntax of commands is extended to include *annotations*. *Sequential* proofs, and all proof rules, do not deal with stability. Combined with the use of syntax-directed proof rules, this design allows for partial automation of sequential proofs. Mechanized sequential proofs are far from easy to perform. Being able to focus on these proofs in isolation is therefore an advantage in terms of proof engineering.

Naturally, we do not claim that an RG proof should be conducted without concern for interference. All predicates should indeed be stated with the right rely conditions in mind. However, the proof system aims at decoupling sequential and concurrent reasoning. Its first layer is a Hoare-like system, without relies or guarantees. A second layer handles interferences: proof obligations about relies, guarantees and stability checks of annotations.

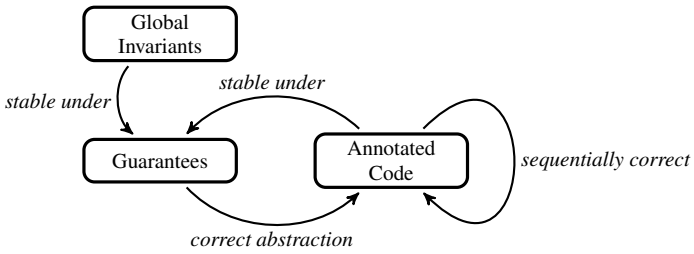

Figure 5: Architecture of an RG proof in our proof system. Specification of code (boxes) is a three-tier system (i) global invariants of the whole program, (ii) guarantees, modelling the action of threads on the shared state, and (iii) thread code annotations, similar to Hoare-logic annotations. We indicate the associated proof obligations with arrows.

To avoid polluting programs with routine annotations, such as global invariants, the first layer of the system *assumes* that such invariants hold, and the second layer requires separate proofs of their invariance as stability checks. Once again, this separates proof concerns. We argue that this is instrumental in designing a tractable workflow. Thanks to this separation, complex invariants can be progressively refined as understanding of the algorithm grows.

Before covering the presentation of the system itself, we explain our design choices by describing the general structure of a rely-guarantee proof. An RG proof in our system can be schematically represented as three interdependent components requiring proof obligations (see Figure 5).

First, the code of each thread is modeled through its guarantees, and global invariants shall be proved stable under any of those guarantees. The code of each thread is annotated, and those annotations must be proved (i) to hold sequentially and (ii) assuming the global invariants, stable under the other threads guarantees: the rely condition is defined as the union of the guarantees of the other threads. Finally, for each thread, we need to prove that its guarantee is indeed an abstraction of the operational semantics of its code.

This structure of the development is different from traditional rely-guarantee systems, where stability conditions are entangled with sequential reasoning by baking them directly into the proof rules. In contrast, we move this concern away from the proof rules, as is described in subsection 3.5.

3.2 Annotations

For annotations we use a shallow embedding into Coq. Annotations are of type either `pred` ≜ `gstate→lenv→Prop`, or `gpred` ≜ `gstate→Prop` when they deal with the global state only. Typically, the global invariant of the collector is of type `gpred`. We also define the usual logical connectives on `pred` and `gpred` with the expected meaning. Conjunction is written `A⩓B` and implication is written `A⟶B`. Annotations of type `gpred` are automatically cast into `pred` when needed.

The syntax presented in section 2 is extended with annotations. We use the informal notation `P`@c for a command c annotated with `P`, and we omit the case of iterators for conciseness. Elementary commands that do not utilize the global state do not need annotations. Commands accessing memory (e.g. field loads and updates, global loads and stores, and

mark-buffer operations) need an extra argument of type `pred`, representing the pre-condition. This is also the case for loops, annotated with a loop-invariant, and atomic blocks, whose body may affect the global state. The semantics of RTIR completely ignores annotations, they are only relevant for proofs.

3.3 Sequential Layer

We start by defining the following predicate, `I ⊨ t: ⟨P⟩ c ⟨Q⟩` which corresponds to the validity of a sequential Hoare tuple, with respect to the big-step operational semantics of commands. This semantic judgment asserts that, for thread `t`, if command `c` runs in a state satisfying precondition `P`, and if the execution terminates, the final state must satisfy post-condition `Q` under the assumption that the global predicate `I` is an invariant. Formally :

```
I ⊨ t: ⟨P⟩ c ⟨Q⟩ ≜
∀ gs1 le1 gs2 le2,
   P gs1 le1 ∧ I gs1 ∧ I gs2 ∧ ((gs1,le1) | c) ⇓t (gs2,le2) ⇒Q gs2 le2.
```

Proving that `I` is indeed invariant of the interleaving semantics is done separately. First-layer logic judgments for commands are of the form `I ⊢ t: ⟨P⟩ c ⟨Q⟩`. For basic commands which do not require annotations and simple command compositions (sequence, non-deterministic choice and loops). The proof rules follow the traditional weakest-precondition style. This can be seen in the following rules:

$$\frac{}{\texttt{I} \vdash \texttt{t:}\ \langle \texttt{P} \rangle\ \texttt{skip}\ \langle \texttt{P} \rangle} \qquad \frac{\texttt{I} \vdash \texttt{t:}\ \langle \texttt{P} \rangle\, c_1\, \langle \texttt{R} \rangle \quad \texttt{I} \vdash \texttt{t:}\ \langle \texttt{R} \rangle\, c_2\, \langle \texttt{Q} \rangle}{\texttt{I} \vdash \texttt{t:}\ \langle \texttt{P} \rangle\, c_1; c_2\, \langle \texttt{Q} \rangle} \qquad \frac{\texttt{I} \vdash \texttt{t:}\ \langle \texttt{P1} \rangle\, c_1 \langle \texttt{Q} \rangle \quad \texttt{I} \vdash \texttt{t:}\ \langle \texttt{P2} \rangle\, c_2 \langle \texttt{Q} \rangle}{\texttt{I} \vdash \texttt{t:}\ \langle \texttt{P1} \wedge\!\!\!\wedge \texttt{P2} \rangle\, c_1 \oplus c_2\, \langle \texttt{Q} \rangle}$$

On the other hand, commands that require annotations directly embed the semantic judgment `I ⊨ t: ⟨P⟩ c ⟨Q⟩` as a proof obligation. For instance:

$$\frac{\texttt{I} \vDash \texttt{t:}\ \langle \texttt{P} \rangle\ \texttt{P@}X = e\ \langle \texttt{Q} \rangle}{\texttt{I} \vdash \texttt{t:}\ \langle \texttt{P} \rangle\ \texttt{P@}X = e\ \langle \texttt{Q} \rangle} \qquad \frac{\texttt{I} \vDash \texttt{t:}\ \langle \texttt{P} \rangle\, c\, \langle \texttt{Q} \rangle}{\texttt{I} \vdash \texttt{t:}\ \langle \texttt{P} \rangle\ \texttt{P@atomic}\ c\ \langle \texttt{Q} \rangle}$$

3.4 Interference Layer

This layer takes into account threads interference with a given command, handling the validity of guarantees and the stability of program annotations *w.r.t.* the context. This can be seen in the definition of a valid RG tuple:

```
Record RGt (t:tid) (R:rg) (G:list rg) (I:gpred) (P Q:pred) (c:cmd) := {
  RGt_hoare:     I ⊢ t: ⟨P⟩ c ⟨Q⟩
; RGt_stable:    stable I P R ∧ stable I Q R ∧ AllStable I c R
; RGt_guarantee: AllRespectGuarantee t I c G }.
```

The type `rg ≜ gstate→gstate→Prop` defines relies and guarantees as binary relations between global states. In our development, we build them from annotated commands. For a command `P@c`, the associated `rg` is defined by running the big-step operational semantics of `c` from a pre-state satisfying `P` to a post-state. We explain in section 5 how our proof methodology benefits from this definition. More precisely, we refer to the interference an atomic command guarded by an annotation may cause to the environment as its *action*.

```
Definition action (t: tid) (ann_cmd:pred*comm) : gstate → gstate → Prop :=
  let '(P,c) := ann_cmd in
  fun g1 g2 ⇒ ∃ l1 l2, P g1 l1 ∧ ((g1,l1) | c) ⇓t (g2,l2).
```

Predicate `stable` defines the stability of a `pred` *w.r.t.* a rely, given some invariant:

```
Definition stable (I:gpred) (H:pred) (R:rg) : Prop := ∀ gs1 gs2 l,
  I gs1 ∧ H gs1 l ∧ R gs1 gs2 ∧ I gs2 → H gs2 l.
```

Predicate `AllStable` then builds the conjunction of the stability conditions for all assertions syntactically appearing therein. It is formally defined as a fixpoint constructing a Coq predicate.

```
Fixpoint AllStable (I:gpred) (c:cmd) (R:rg) (S: Prop) : Prop :=
  match c with
  | P@i ⇒ stable I P R ∧ S
  | P@loop(c) ⇒ stable I P R ∧ AllStable I c R S
  | c1;c2 | c1 ⊕ c2 ⇒ AllStable I c1 R (AllStable I c2 R S)
  [...]
  end.
```

The validity of the guarantee of a command (predicate `AllRespectGuarantee`) follows the same principle, this time accumulating proof obligations that all elementary effects of the command are reflected by an elementary guarantee in the list `G`.

3.5 Program RG specification

The RG specification of a program `p` is defined as a record considering guarantees `G` and pre- and post-conditions `P` and `Q` for all threads. Formally:

```
Record RGt_prog (G:tid → rg) (I:gpred) (P Q:tid → pred) (p:program) := {
  RGp_t:∀ t ∈ (threads p), RGt t (Rely G t) (G t) I (P t) (Q t) (cmd t p)
; RGp_I:∀ t, stable TTrue I (G t) }.
```

The proof obligation `RGp_t` requires that each thread's command is proved valid. It is worth noting that only guarantees need to be considered: for each thread, we build its rely from other threads' guarantees (`Rely G t`). This significantly reduces redundancies in specifications. Second, the obligation `RGp_I` requires that `I` is invariant. We encode this as a stability condition under the union of all threads' guarantees, assuming the trivial invariant `TTrue` ≜ `(fun _ _ ⇒True)`. Indeed, as all threads' code satisfy their guarantees, this is enough to prove that the global invariant `I` is preserved by any number of program steps.

3.6 Reasoning about Iterators

As expected, iterators are more involved. We illustrate their treatment on `foreach`; other iterators are similar. Recall that `foreach` iterates on a list of data of type `A`, morally representing a loop. Hence, its proof involves a loop invariant, predicated over the visited elements of the list. Predicates annotating `foreach` are indexed by a list of visited elements. And, as the loop body may include annotations about visited elements, we also index it by a list of visited elements and a current element. Summing up, the syntax of `foreach`, extended with annotations is `P@foreach` (*x in* `l`) `do c od` where annotation `P` has type `list A → pred`, and

`c` has type `list A → A → cmd`. The associated proof rule is:

$$\frac{\begin{array}{l}\texttt{∀ a seen, prefix (seen++[a]) l →}\\ \qquad\texttt{I ⊢ t: ⟨P seen⟩ (c seen a) ⟨P (seen++[a])⟩}\\ \texttt{(P l)⩓I ⟶ Q}\end{array}}{\texttt{I ⊢ t: ⟨P nil⟩ P@foreach (}x\ \textit{in}\ \texttt{l) do c od⟨Q⟩}}$$

The first premise amounts to proving a valid tuple whose pre- and post-conditions are adjusted to the list of already visited elements. The second premise requires pre-condition `P` applied to the whole list of elements to entail the post-condition of the iterator itself. We define a more general rule in Coq, to get an induction principle usable to prove the soundness of the logic.

3.7 Soundness of the logic

Soundness states that invariant `I` holds in every state reachable from a well-formed initial state – which must satisfy `I` by construction – through the small-step semantics mentioned in section 2. Formally:

```
Hypothesis init_wf : ∀ tsi gsi, init_state p (tsi,gsi) →
     RGt_prog G I P p Q                          (* program RG spec *)
   ∧ (∀ t c le, tsi(t) = Some(c, le) → P t gsi le) (* pre-conds. hold *)
   ∧ I gsi.                                      (* I holds initially *)
Theorem soundness : ∀ ts gs, reachable init_state p (ts,gs) → I gs.
```

The proof of this theorem relies on an auxiliary proof system, proved equivalent to the one presented earlier. The auxiliary system reuses the same components, but proof rules now require to prove everything in situ: invariant, pre- and post-conditions, stability of annotations, and validity of guarantees. For instance, compare the rule for instruction $X = e$ in the previous system (left) with the proof rule of the auxiliary system (right):

$$\frac{\texttt{I ⊨ t: ⟨P⟩ P@}X = e\ \texttt{⟨Q⟩}}{\texttt{I ⊢ t: ⟨P⟩ P@}X = e\ \texttt{⟨Q⟩}} \qquad \frac{\begin{array}{c}\texttt{TTrue ⊨ t: ⟨P⩓I⟩ P@}X = e\ \texttt{⟨Q⩓I⟩}\\ \texttt{stable TTrue (P⩓I) G}\quad \texttt{stable TTrue (Q⩓I) G}\\ \texttt{RespectGuarantee t I G (P@}X = e\texttt{)}\end{array}}{\texttt{R, G, I ⊢ t: ⟨P⟩ P@}X = e\ \texttt{⟨Q⟩}}$$

This auxiliary system is very close to classic RG systems [18,40]. Its verbosity makes it easier to reason about during the soundness proof.

To prove the above semantic soundness theorem, we proceed as follows. We prove that, in all reachable execution states, any running thread is in fact running a piece of code that conforms to `RGt`. Hence, after each execution step, we need to update the map `P` of threads pre-conditions. The invariance of the global invariant `I` follows from the fact that, in each proof rule (see the above example rule on the right) of the auxiliary system, `I` is part of the pre- and post-conditions, and these are stable against any step of the rely and the guarantee of the stepping thread.

4 The Concurrent Garbage Collector

We now describe our implementation of the concurrent collector and its associated correctness theorem. The algorithm is based on [8], a variant of the well known concurrent *mark-and-sweep* algorithm due to Doligez, Leroy and Gonthier [7,6].

4.1 Main Theorem

Intuitively, we show that the collector thread never reclaims memory that could potentially be used by mutators. To do this, we program the collector and the mutators in RTIR, and prove that their parallel composition preserves an invariant on global execution states, using our program logic.

The particularity of mutators is that they participate in the bookkeeping required for the collection to be correct. In practice, bookkeeping code is injected in client code by the compiler. Here, we consider a *Most General Client* (MGC) representing a collector thread composed with an arbitrary number of mutators with identifiers in Mut, each running relevant injected pieces of code.[1]

$$\begin{aligned} \texttt{mutator} \triangleq\ & \mathsf{loop}\,(\ \texttt{update}(\texttt{x},\texttt{f},\texttt{r}) \oplus \texttt{x.f} = \texttt{n} \oplus\ \texttt{x} = \texttt{y.f} \\ & \qquad \oplus\ \texttt{Alloc}() \oplus\ \texttt{cooperate}() \oplus\ \texttt{changeRoots}()) \\ \texttt{mgc} \triangleq\ & \texttt{collector} \parallel \texttt{mutator} \parallel \ldots \parallel \texttt{mutator} \end{aligned}$$

The MGC can update values in memory – directly for numerical values, through an instrumented operation for references –, perform loads, instrumented allocation, execute a cooperation subroutine and simulate the tracking of roots involved by local move operations. We detail these notions through subsection 4.2. User global variables are disallowed in the current development, but adding them would be straightforward: the collector would simply have to publish them itself while it waits for the mutators to publish their roots.

Recall that the special global variable freelist holds a pool of unused references. Hence, upon allocation, a reference is fetched from the freelist. Symmetrically, to reclaim an unused object, the collector puts back its reference into the freelist. Our main invariant establishes that in a given state gs, any reference r reachable from any mutator m is not in the freelist, and hence has not been collected.

```
Definition I_correct: gpred :=
  fun gs ⇒ ∀ m r, In m Mut ∧ Reachable_from m gs r → ¬ in_freelist gs r.
```

We can now formulate our main theorem. It uses the predicate reachable_mgc stating that a global state gs can be reached, from a predefined initial state, by the code of the MGC.

```
Theorem gc_sound: ∀ gs, reachable_mgc gs → I_correct gs.
```

The initial state is obtained by a startup phase of the runtime, that carefully initializes intrinsic features of the runtime, and establishes key invariants.

Obviously, proving this theorem would be impossible without the aid of other intermediate invariants. In the sequel we explain the important aspects of the implementation, and a few salient auxiliary invariants. Describing the algorithm and code in full details is out of the scope of this paper. We refer the reader to [8] and to the formal proof [42] for details.

4.2 High-level Principles of the Algorithm

Our GC is of the *mark and sweep* family: the heap is traversed, marking objects that are presumably alive, i.e. reachable from mutators' local variables, henceforth called *roots*. Once

[1] We present a simplified pseudo-code of the MGC, with variable x, field f, and value v assumed nondeterministically chosen from the thread environment. The actual definition in Coq is an operational characterization of this thread system.

```
// collector ::=
while (true) do
  atomic  // ghost
    stage[C] = CLEAR
    phantom_flipped = 0
  atomic  // linearizable[4]
    foreachObject o do
      if !(isFree(o)) then
        o.color = WHITE
    od
    phantom_flipped = 1
  handshake()  // SYNCH1
  handshake()  // SYNCH2
  stage[C] = TRACING
  handshake()  // ASYNCH
  trace()
  stage[C] = SWEEPING
  sweep()
  stage[C] = RESTING
od
```

Listing 1: Collector

```
// handshake() ::=
phantom_hdsk = 1
phase[C] = phase[C] + 1 mod 3
foreach (m in Mut) do
  repeat skip
  until phase[m]==phase[C]
od
phantom_hdsk = 0
```

Listing 2: Handshake

```
// tid m : cooperate ::=
if phase[m] != phase[C] then
  if phase[C] == ASYNCH then
    foreachRoot (r of m) do
      markGrey(buffer[m], r)
    od
  phase[m] = phase[C]
```

Listing 3: Cooperate

```
// tid m : update(x,f,v) ::=
if (phase[m] != ASYNCH
    || stage[C] == TRACING) then
  old = x.f
  markGrey(buffer[m],old)
  markGrey(buffer[m],v)
x.f = v
```

Listing 4: Write Barrier

```
// markGrey(buffer,x) ::=
if (x != NULL
    && x.color != BLACK) then
     buffer.push(x)
```

Listing 5: MarkGrey

the marking procedure finishes, the sweeping procedure reclaims objects detected as not reachable by putting them back in the freelist.

The marking conventions for denoting object reachability follow the tricolor convention [5]. Color `WHITE` is used for objects not yet seen. `GREY` is used for visited, hence presumably live objects, whose children (through fields accesses) have not yet been visited. `BLACK` is used for visited objects whose children have all been visited. In our implementation, colors `WHITE` and `BLACK` are implemented with numerical constants. We explain the encoding of `GREY` later. The heap traversal (marking) procedure is called *tracing*, and completes once no `GREY` objects remain.

Extra care is required to cope with the concurrent execution of mutators: they could modify the object graph at any point, and thus invalidate the properties of the coloring. In particular, mutators are responsible for publishing their own roots by marking them as `GREY` before tracing begins. This is the goal of the `cooperate` procedure. Similarly, object field update should not break color-related reachability invariants during tracing. This is the goal of the *write-barriers*, implemented by the `update` procedure. Finally, the right color should be assigned to newly allocated objects. For space reasons, we elide the details of the `alloc` procedure that we have implemented, and refer to [42,8] for details.

All these subtle procedures, run by the collector and the mutators, are orchestrated using the global variable `stage[C]`, which encodes for the various stages of the collection cycle (including the tracing and sweeping), and the global variables `phase[m]` – one for each mutator – and `phase[C]` – one for the collector – to coordinate mutators with the collector. A diagrammatic representation of a collection cycle is shown in Figure 6, gathering all previously mentioned ingredients. We will refer to it in more detail below.

4.3 RrIR Implementation and Main Invariants

Let us now describe the implementation. Code snippets in RrIR use a simplified syntax from the one we presented above for space and readability reasons.

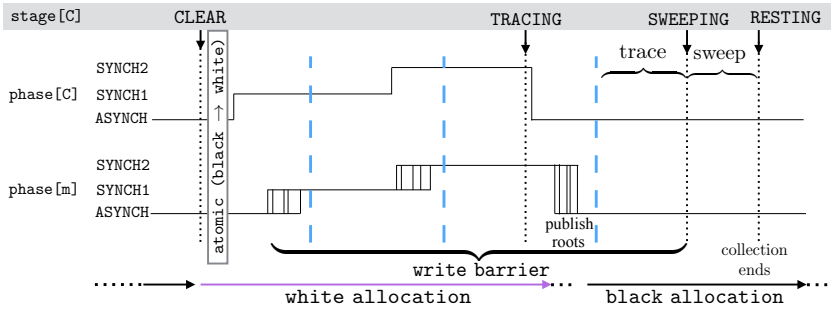

Figure 6: Timeline of a collection cycle. All mutators are coalesced into the bottom line, and the collector is shown on the top line. Dotted lines represent the GC start of a new stage, and dashed lines represent the end of a phase change (handshake).

Stage and Phase Protocol. The code of the collector is presented in Listing 1. For the moment, let us concentrate only on the calls to the `handshake()` procedure (Listing 2), and its counterpart `cooperate()` (Listing 3) executed by the mutators. A collection cycle is structured using four stages: `CLEAR`, `TRACING`, `SWEEPING` and `RESTING`. The current stage is written by the collector to a global variable `stage[C]`. This global variable allows mutators to coordinate with the collector at a coarse level. At a finer level, a handshake mechanism is required, and the status of each thread, the mutators and the collector, is tracked with a `phase` variable, with values ranging over `ASYNCH`, `SYNCH1` or `SYNCH2`. Each phase is encoded with a dedicated integer between 0 and 2. Instead of presenting a detailed description to justify these phases, let us point out that the original algorithm of [7] used only two phases, which was later discovered to be incorrect. A new phase was added to correct it in [6].

We concentrate now on the horizontal lines of Figure 6, showing the evolution of `phase[C]`, as well as the aggregated representation of all the `phase[m]` variables of mutators. Each phase starts by the collector modifying the `phase[C]` variable (second line of Listing 2). Mutators query it (first line of Listing 3), to acknowledge possible changes, in which case mutators *respond* by updating their own `phase[m]` variable (last line of Listing 3). When the collector acknowledges that all mutators have updated `phase[m]`, the phase transition is completed (dashed line in Figure 6). Importantly, `phase[C]` and `phase[m]` are subject to race conditions. We also point out that threads never stop their execution while executing `cooperate`.

An important invariant relating the phases of the collector and the mutators is that any mutator's phase is at most one step behind the collector's phase.

```
Definition I_synch : gpred := fun gs ⇒
∀ m, In m Mut → phase[C]gs = phase[m]gs ∨ phase[C]gs=(phase[m]gs+1) mod 3.
```

Buffers and `GREY`. Objects are marked `GREY` with the `markGrey` procedure (Listing 5) either when mutators publish their roots (Listing 3) or during write barriers (Listing 4). Each mutator owns a `buffer[m]` abstract data structure, in which it adds references to be traced. Hence, `buffer[m]` serves as an interface between mutators and the collector to mark objects as `GREY`. In other words, an object is considered `GREY` if it is present in any `buffer` and its color field is `WHITE`. In this sense, `GREY` is a convention rather than a true color.

```
1  // trace() ::=
2  all_empty = false
3  while (!all_empty) do
4    atomic // ghost code
5      foreach (m in Mut) do
6        phantom_buffer[m].copy(buffer[m])
7      od
8    all_empty = true
9    foreach (m in Mut) do
10     is_empty = buffer[m].isEmpty()
11     while (!is_empty) do
12       all_empty = false
13       x = buffer[m].top()
14       if (x.color == WHITE) then
15           buffer[C].push(x)
16           buffer[m].pop()
17       else buffer[m].pop()
18       is_empty = buffer[m].isEmpty()
19     od
20   od
21
22   while (!buffer[C].isEmpty()) do
23     all_empty = false
24     ob = buffer[C].top()
25     if (ob.color == WHITE) then
26       foreachField (f of ob) do
27         if (ob.f!=NULL
28             && ob.f.color==WHITE) then
29           buffer[C].push(ob.f)
30       od
31       ob.color = BLACK
32     buffer[C].pop()
33   od
34 od
35
36
37 // sweep() ::=
38 foreachObject o do
39   if (!isFree(o) && o.color == WHITE) then
40     free(o)
41 od
```

Listing 6: Trace and Sweep (Collector)

Write barriers. Their code is shown in Listing 4. The barrier conditionally either directly updates field `f` (fast-path) or marks two objects as `GREY` (slow-path).[2] Notice that the slow-path is executed only when the collector is ready to start tracing, and not after it started sweeping. The code of write barriers is intrinsically racy since the client code itself might contain races at `f`; moreover, accesses to the `buffer` data structures are not protected by synchronization between mutators and the collector. Finally, note that the order in which the `markGrey` operations are performed in the write barrier is critical to the GC correctness.

Trace This is the most challenging code to verify, and its verification by means of program logics would be remarkably hard without some of the design choices of RTIR, and our proof methodology. The `trace` procedure (Listing 6) traverses the object graph starting from `GREY` objects. More precisely, the collector visits each of the mutators `buffer[m]` in the `foreach` loop at Line 9, transferring their contents into its own `buffer[C]`. If the collector sees empty buffers for all mutators, tracing ends. Otherwise, it traverses the graph starting from objects in `buffer[C]`, and marking `BLACK` objects whose children have been seen.

Regarding the complexity of the code, we emphasize that it contains three nested loops, a number of `foreach` constructs, and heavily uses the buffer abstract data structures. Moreover, it exhibits races in all threads (through write barriers and buffer operations) since it traverses the object graph, while mutators concurrently modify it.

An important invariant establishes that during the tracing phase, any `WHITE` object that is alive must be reachable from a `GREY` object, signaling that it still has to be visited. Since another invariant, `I_black_to_white`, states that any path from a `BLACK` object to a `WHITE` object goes through a `GREY` object, this translates to the property that all objects reachable from the roots are either `BLACK`, or reachable from a `GREY` one.

```
Definition I_trace_grey_reach_white : gpred := fun gs ⇒ ∀ m r,
stage[C]gs ≠ CLEAR ∧ In m Mut ∧ phase[m]gs=ASYNCH ∧ Reachable_from m gs r→
 Black gs r ∨ (∃ r0, Grey Mut gs r0 ∧ reachable gs r0 r).
```

When this code terminates, we are able to prove that: (i) there are no more `GREY` objects, (ii) all objects reachable from the mutators roots are `BLACK`, and consequently (iii) there are

[2] The write barrier in [8] avoids marking `old` in some cases. We drop this optimization.

no WHITE objects reachable from any of the mutators roots. Property (i), namely that all buffers are *simultaneously* empty at the end of tracing (Listing 6, Line 35), is particularly difficult to prove, given the write barriers executed concurrently by mutators. We prove that this property is established at Line 4 of the last iteration of the outer while loop as follows. We first prove that, at Line 4, `buffer[C]` is always empty. We use ghost variables `phantom_buffer[m]` to take the snapshot of mutators' buffers at Line 4. Mutators can only push on their buffers, so, in a given iteration of the outer loop, if a mutator buffer is empty, so was its ghost counterpart during the same iteration. In the last iteration, all buffers are witnessed empty, one at a time. But this implies that all phantom buckets are simultaneously empty at Line 8. This, in turn, implies that all buffers are, this time *simultaneously*, empty at Line 4. This property remains true until Line 35: it is both stable under mutators' guarantees, and preserved by the inner loop. Indeed, if all buffers are empty (there are no GREY objects), the above invariant `I_trace_grey_reach_white` implies that both the old and new objects that `markGrey` could push on a buffer are in fact BLACK, and thus not pushed on any buffer (Listing 5). As a consequence, no reference is pushed on the collector's buffer (Line 15).

Sweep The sweep phase (Listing 6) recycles all the objects that remain WHITE after TRACING. This is the only place where instruction `free` is ever used. Note that this code is also non-blocking. The key property shown above is that during sweeping, no GREY objects remains:

```
Definition I_sweep_no_grey : gpred := fun gs ⇒
  (stage[C]gs = SWEEPING ∨ stage[C]gs = RESTING) → ∀ r, ¬ Grey Mut gs r.
```

This invariant, with `I_trace_grey_reach_white` above, implies that no WHITE object is reachable from any thread-local variable.

5 Proof methodology

Mechanizing such a sizable proof raises methodological concerns. While the proof system of section 3 separates proof concerns between sequential reasoning and stability checks, we deal here with the intrinsic complexity of the proof and its scalability.

First, stating up-front the right set of invariants, guarantees, and assertions is unrealistic for such a proof. To tackle this issue, we group invariants related to distinct aspects, e.g. the phase protocol or coloring invariants. To reflect this structure in our proof, and avoid constant refactoring of proof scripts, we designed an incremental workflow.

Second, we must deal with the quantity of proof obligations. For the GC code, proof obligation `RGp_I` involves 18 invariants, each of which must be proved stable under 17 guarantees, thus requiring 306 stability proof obligations. On top of this, proof obligation `RGt_stable` adds more than 60 annotated lines of code, each bearing several predicates, that must be proved stable under significant subsets of the 17 guarantees. This quickly becomes intractable without a disciplined methodology and some automation.

5.1 Workflow

An important aspect of the proof is that the synchronization protocol between threads can be proved correct in isolation, without any concern about graph reachability or coloring invariants. By combining our incremental methodology with an RG proof that follows the architecture described in Figure 5, we are able to establish the validity of invariant `synch_protocol`

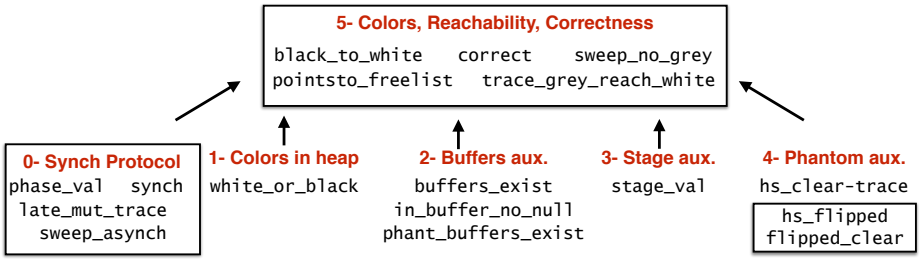

Figure 7: Main Invariants of the GC. Numbers are timestamps in the incremental proof methodology. Dependencies are shown with boxes (inter-dependency) and arrows.

that describes the protocol on a partially annotated code, modeled by a set of guarantees with weakly constrained preconditions.

However, trying to enhance this first result with additional invariants is highly impractical: for the new invariants to hold, the guarantees will have to be further refined to take into account new subtleties about the code. Once the `synch_protocol` has been proved stable with respect to the previous definition of the guarantees, all proofs of stability can potentially break. Nevertheless, all stability lemmas we established should still hold since we only gather more information about the execution, hence further restricting the possible interferences. In practice however, mechanized proof scripts are extremely fragile, and constant refactoring of proofs is a daunting and time consuming task. We therefore formalize this intuition to build a methodology allowing us to actually work incrementally.

To do so, we organize the invariants in groups, as illustrated on Figure 7. These groups establish respectively, from bottom to top and left to right: the synchronization of the threads during the collection cycle; that cells are only colored white or black; the well-formedness of the buffers; the possible values taken by the `stage` variable; relating the value of ghost variables introduced to reason; and finally the major graph invariants and the corectness theorem. We refer the reader to our formal development for the exact definition of the invariants.

In each boxed group, invariants are inter-dependent, while arrows indicate a dependency of the target group on the source group. We then use a simple mechanism: invariants and guarantees are indexed by a natural number – intuitively a timestamp of their introduction into the development. When introducing a new increment to an invariant, all invariants with a lower timestamp are therefore not modified. Neither are their proofs, resulting in an incremental, non-destructive methodology. More concretely, each time we add a new timestamp in the incremental proof: (i) we enrich the invariant, refine the guarantees and code annotations; (ii) we prove the new stability proof obligations, for which we can reuse prior stability proofs, and we use automation to discharge as many obligations as possible; (iii) we adapt sequential Hoare proofs, and prove that enriched guarantees are still valid. This workflow proved robust during our development, allowing for an incremental and manageable proof effort. We detail below the first two items of this methodology.

5.2 Incremental proofs

We focus on obligation `RGp_I` from section 3, which requires establishing invariant stability under all threads' guarantees. Let us fix a thread and index both invariant `I` and guar-

antee G by n. The obligation is thus (stable TTrue (I n) (G n)). We want to establish (stable TTrue (I n+1) (G n+1)) by using the already proved (stable TTrue (I n) (G n)) obligation.

Monotonicity of I *and* G. We build (I n+1) as a conjunction of the prior established invariant (I n), and the current level increment: (I n+1) $\triangleq$ (I n) $\curlywedge$ (Ic n+1). Hence, we have that $\forall$n, (I n+1) $\longrightarrow$ (I n). Recall that in our proof system, guarantees are expressed through the effect of a command, under certain hypotheses on the pre-state. At each level, the command will not change – it is effectively executed by the code. The levels are used to refine the hypotheses on the pre-state. Therefore, guarantees are monotonic in the sense that $\forall$n, (G n+1) $\subseteq$ (G n): they are made more precise as the level index increases.

Reuse of Proof of Prior Invariants. We start by proving that the prior invariant (I n) is stable under the refined guarantee (G n+1), i.e. (stable TTrue (I n) (G n+1)). To do so, we reuse our previous proofs at level n and conclude with the following lemma using guarantee monotonicity – below, we abuse notations and use _ as a valid Coq identifier.

```
Lemma stable_refineG: ∀ _ I G1 G2, G2 ⊆ G1 ∧ stable _ I G1 → stable _ I G2.
```

New Invariant Stability. It remains to prove the stability of increment (Ic n+1) under the refined guarantee (G n+1). In simple cases, (stable TTrue (Ic n+1) (G n+1)) is provable independently from prior invariants. In this case, we combine the stability of (Ic n+1) and (I n) into the one of (I n+1) with lemma stable_and below.

```
Lemma stable_and: ∀ _ I1 I2 G, stable _ I1 G ∧ stable _ I2 G → stable _ (I1 ⩓ I2) G.
```

However, the situation is often more involved, requiring prior invariants to prove the stability of (Ic n+1). Formally, we have (stable (I n) (Ic n+1) (G n+1)). We can then combine the stability of (I n) and (Ic n+1) under (G n+1) using the following lemma.

```
Lemma stable_with: ∀ _ I1 I2 G, stable _ I1 G ∧ stable I1 I2 G → stable _ (I1 ⩓ I2) G.
```

Finally, for code annotations to go through a similar indexing treatment, we introduce an indexed conjunctive operator and_at. Predicate (and_at n c A B) *activates* annotation B at index c only if B was introduced at a prior index n. Formally :

```
Definition and_at (n c: nat) (A B: pred): pred :=
if leb n c then fun gs le ⇒ A gs le ∧ B gs le
else A.
```

We then define a function comm_impl computing a sufficient set of proof obligations required to leverage a proof of stability of the code from one iteration index to the next. The proper definition of comm_impl is quite technical, but the intuition behind it is simple. When considering two commands c_1 and c_2, which are expected to be two versions of the same piece of code, comm_impl inductively traverses both commands simultaneously. If any operational mismatch is found, it generates an impossible proof obligation. Otherwise, it compares the code annotations P_1 and P_2 and generates a proof obligation expressing both (i) that P_2 is a strengthening of P_1 and (ii) that P_2 is stable under the current guarantees, assuming the stability of P_1 that we are meant to have previously established. These generated proof obligations are proved correct in the sense that they are sufficient to lift a previous proof of stability of a piece of code:

```
Lemma AllStable_strengthenC: ∀ rt I c1 c2 G S,
  comm_impl rt I c1 c2 G → AllStable rt I c1 G S → AllStable rt I c2 G S.
```

Assisted with dedicated tactics, we recover at the level of annotated code the same benefits we obtained for the stability of invariants.

5.3 Proof Scalability

To deal with the blowup of stability checks, we built a toolkit of structural stability lemmas, and developed some tactic-based partial automation. Structural stability lemmas are meta-properties on the notion of stability. They rely on the logical structure of the predicates to reduce their proof of stability to the stability of predicates which are either simpler to prove stable, or syntactically different but logically equivalent. This allowed us to automatically discharge 186 obligations among the 306 induced by `RGp_I`. The remaining obligations are partially reduced by this automation.

Structural lemmas have two purposes. First, they enable the incremental methodology described above, allowing to leverage stability results from lower levels. Second, they simplify complex stability proof obligations: both annotations and invariants as well as interferences can be structurally split up. Thus, complex arguments are isolated from trivial ones, that are automatically discharged. Similarly, by decomposing annotations and relaxing interferences, we can factor out the stability proofs for annotations that appear several times in the code.

Automation. We developed a set of tactics that simplify stability goals into elementary ones before attempting to solve them. This leads to clearer goals and more tractable proof contexts. The tactics combine structural lemmas with two additional ideas: systematic inversion on guarantee actions – defined operationally using commands – and aggressive rewriting in predicates.

6 Closing the gap toward executability

The methodological approach we took to tackle the proof of the collector in a structured way led us to introduce several non-executable abstractions. Those abstractions, some embedded into RTIR, some into the algorithm itself, have to be refined in order for the GC to be executable. We review through this section the steps required to achieve full executability.

On the language side, two kinds of refinements need to be performed. Probably the most challenging of these is the implementation of the data-structures abstractly supported in the language. Some of the authors already investigated this difficulty by providing a generic Rely-Guarantee-based framework to devise observational refinements of linearizable data-structures [43]. This result has been instantiated to prove correct a compilation pass from RTIR to an identical language, but which manipulates a concrete fine-grained implementation of the mark-buffers as introduced by Domani et al. [8]. Two other components of RTIR need to undergo a similar process. First, the management of roots should be implemented as a data structure. However, although for convenience we track them in the shared state of our development, this data-structure is purely thread-local. This refinement would therefore raise little challenges. Second, the implementation of the `freelist` is significantly harder. At its core, the data-structure is a shared list, typically implemented as a Treiber's stack [38] or a Michael-Scott queue [25]. On top of it, fragmentation of the memory needs to be taken into account to achieve satisfying performance. To simplify things, it should nonetheless be possible to handle the management of the fragmentation at a thread local level so as to

restrict the use of the shared data-structure to its simplest. Refining the freelist therefore involves both a careful design of the algorithm, and some non-trivial fine-grained reasoning.

A second feature of the language which requires attention is the blocking aspects of the semantics. As explained in Section 2.2.6, removing all blocking states from RTIR involves several orthogonal concerns. First, dynamic type checking side-conditions in the semantics should be proved to always succeed, through the use of a sound static type system, type-checking the input program of mutators as well as the injected code. Second, the absence of temporal memory errors, such as the dereferencing of dangling pointers or double-free errors, are encoded in the language through blocking states. The functional correctness of the GC we establish in this paper precisely proves their absence in presence of the GC, and hence is the tool needed to remove these blocking states further down in the compilation chain. Note that this step conceals a couple of prerequisites: given the concrete compilation pass that injects the GC into a client, one needs to transfer the invariant `I_correct` from the `MGC` to the client. However, we believe this to raise no difficulty. Finally, blocking states related to the handling of the `NULL` pointer constitute an orthogonal matter.

On the algorithmic side, three elements are relevant to executability. First, ghost code should be cleaned up. Doing so is essentially administrative since ghost variables are exclusively written to, but never read, and hence do not interfere with the GC's observational behavior. Second, the bitflip performed at the beginning of each cycle by the collector is modeled as a loop, explicitly setting the `color` fields of all references to `WHITE`. Refining this abstraction is not expected to raise difficulties. Last, the instrumented allocation routine is wrapped inside an atomic block in our current development. We leave its refinement as a further work. This choice of design, that is shared by Gammie et al. [10], is motivated by the fact that an efficient implementation of the `freelist` might not be linearizable. A definitive proof of the allocation subroutine would therefore only make sense once the choice of representation for the `freelist` is made.

Finally, from a broader perspective, (verified) executability of the GC requires RTIR to be injected into a suitable verified compiler. This perspective is significantly more ambitious. On the language side, in our current RTIR design, the loop body of iterators are shallowly-embedded Coq functions. This eases the reasoning on their algorithmic behavior but will require extra effort to compile RTIR into a lower intermediate representation. Indeed, such loop bodies can not be directly pattern-matched by a Gallina function. We believe that going from a shallow representation to a deep representation could be achieved manually by requiring the RTIR programmer to provide both versions, and prove their equivalence. According to our preliminary experiments, such proofs could be reasonably automated. It is also worth noticing that no out-of-the-box adequate candidate currently exists for a verified compiler. Indeed, such a verified compiler should naturally handle concurrency, a restriction which essentially reduces the list to a single candidate, CompCertTSO [36]. However, CompCertTSO, as an extension of CompCert, is designed to handle the C language. The compiler hence manipulates intermediate representations carrying very little memory-safety-related invariants. For the GC to be able to iterate over the memory, one should always be able to interpret the structure of an object pointed to by a reference, and hence requires a memory model maintaining higher-level information than those offered by CompCert's memory model. CompCertTSO stresses out another major perspective for our work. In order to be able to soundly compile down to architectures, we first need to extend our result to support weak memory models, the Total Store Order (TSO) one in particular.

7 Related Work

The literature on garbage collection is vast. We refer the reader to [19] for a comprehensive and up-to-date presentation of garbage collection techniques. The starting point for our work is [8], a state-of-the-art on-the-fly concurrent GC based on the earlier DLG algorithm [7,6]. Many of the invariants we prove are inspired by those of [6].

Verified sequential garbage collectors. Garbage collectors have always been a paradigmatic verification challenge. Many prior efforts have tackled various instances of the sequential case. A similar copying collector algorithm has been proved correct both in the CakeML compiler [26] and the Milawa prover [2]. Some impressive projects providing end-to-end verified specialized operating systems include a verified garbage collector, such as Ironclad Apps [13] and Verve [41].

Ericsson et al. [34] have recently equipped the CakeML compiler with the first verified generational copying garbage collector. Their proof is of particular interest for two reasons. First, they are embedded in the CakeML compiler and hence fully executable. Second, they managed to structure the proof quite elegantly. A partial collection of the heap over a generation is proved correct by being simply interpreted as a collection over a smaller heap from which exiting pointers are ignored.

Hawblitzel and Petrank [14] introduced an interesting different line of work in 2009. They rely on the Boogie verification condition generator [20] in combination with the Z3 SAT/SMT solver [3] to verify down to assembly two sequential garbage collectors, an elementary mark-and-sweep algorithm and a Cheney copying algorithm. The approach consists of implementing the algorithms in x86 assembly, and heavily annotating the code. Boogie and Z3 then take care of generating and discharging the corresponding proof obligations.

Garbage collectors have also been a playground for separation logic in recent years, the logic easing the reasoning about programs involving pointers by allowing for local reasoning. Torp-Smith et al. [37] use it to verify a copying garbage collector. McCreight et al. [24] extend CompCert with a new intermediate language, GCminor, equipped with memory management primitives and providing a target of compilation for managed languages. Using separation logic, they subsequently partially prove a Cheney copying collector. In 2010, McCreight et al. [23] prove a machine-level implementation of a variety of standard garbage collector implementations.

Verified concurrent garbage collectors. The previously cited works deal with sequential collectors. The first argument of correctness of a concurrent collector were introduced by Dijkstra et al. [5] with the three colors abstraction. The proof is however not fully formal. The first mechanized proof was presented by Gonthier [11], and conducted in the TLP system. Unlike ours, Gonthier's proof rests on an abstract encoding of the algorithm. The work of Havelund [12] in PVS also verifies an abstract algorithm. Pavlovic et al. [29] propose a different approach, synthesizing a more realistic concurrent collector from a simpler, abstract, initial implementation. While the approach is certainly appealing, they do not manage to reach algorithms as fine-grained as the one we consider here.

In 2015, Gammie et al. [10] verified an on-the-fly algorithm close to ours. Similarly to Gonthier's work, they do not perform the proof over the code itself, but over an abstract model of the algorithm. Their proof is monolithic, following an Owicki-Gries approach. However they take into account the Total Store Order (TSO) memory model, exhibited by x86 processors. On the other hand, they degrade Domani et al.'s algorithm by enforcing unnecessary synchronizations to ease reasoning. In contrast, our algorithm clearly exhibits

some TSO relaxations, and some traces will exhibit read-write re-orderings. We would have to prove that these reorderings do not break our invariants. We believe we could prove that most of our annotations are "stable" under such reorderings and prove a meta-theorem showing that this kind of stability implies correctness under TSO semantics.

Some extensions of separation logic [33] to concurrency [1,27] could be leveraged to tackle the proof of concurrent garbage collectors. However, to the best of our knowledge, the only successful attempt to date is the work by Liang et al. [21,22]. They introduce a Rely-Guarantee-based Simulation (RGSim), embedding separation logic assertions, for compositional verification of concurrent transformations. Their work can be thought of as a syntactic proof system to build simulations between concurrent programs. They use this system to prove a mostly-concurrent garbage collector in a pleasantly concise way. While the garbage collector they handle is simpler than ours and its proof is not mechanized (only the meta-theory of the proof system is), they offer a very promising line of research.

By sidestepping any modeling in our formal development and proving directly the implementation in RTIR, we argue that our work constitutes the closest-to-executable verified on-the-fly garbage collector to date.

Mechanized concurrent program logics. In [32] an RG logic for a simple imperative concurrent language is formalized and proved sound in Isabelle/HOL. In contrast, our program logic is customized for runtime system implementations, and therefore supports local and global environments, references, iterators, etc. Also, the proof rules of [32] mix sequential reasoning with side conditions about stability and guarantee checks. We decouple these aspects and avoid redundancies by extracting relies from the guarantees of the context. Other approaches to the mechanized verification of concurrent code are [9,39,22,35] to mention but a few. These works are mostly concerned with concurrent data structure correctness, whereas we are concerned with the implementation of a runtime system.

8 Conclusion

This paper presents the mechanized proof of an emblematic challenge in program verification: an on-the-fly concurrent garbage collector. Overcoming this challenge requires a number of methodological milestones. We follow a programming language-based approach: a well-chosen intermediate representation, a companion program logic, and a dedicated proof workflow. RTIR strikes a balance between low-level features for the expression of efficient concurrent code, and high-level features which remove the burden of dealing with low-level details in the proofs. Our program logic is inspired by Rely-Guarantee, a milestone in concurrency proof techniques, but one that had yet not been used for the mechanized verification of garbage collectors. Our incremental proof workflow, combined with specific and efficient tool support via Coq tactics, is efficient and flexible enough for such a verification challenge.

There are two major avenues for future work. The first one is pragmatic, and concerns the embedding of our work in a verified compiler tool chain. Using our theorem about the most-general client, we can build a refinement proof between an IR with implicit memory management and RTIR. We then need to have a fully executable version of the GC. This would require cleaning up ghost code, coding iterators as low-level macros, and implementing abstract concurrent data structures supported by RTIR. The two first tasks are essentially administrative. The third task is more challenging, requiring us to formally prove an atomicity refinement result for linearizable, fine-grained data-structures. To that end, we have developed the meta-theory in [43].

The second direction of future work is methodological. Many of our stability proof obligations are currently discharged using basic typing information: stability holds because assertions or invariants talk about memory that is untouched by some program fragments. An alternative to our basic well-typedness conditions would be to resort on techniques like Separation Logic [33]. In particular, Vafeiadis proposes in [40] a method combining RG and Separation Logic. It would be interesting to see how these techniques could be leveraged in our proof.

Acknowledgements We thank the anonymous reviewers and Peter Gammie for their thorough comments and suggestions on how to improve the final version of the paper. We also thank Vincent Laporte for his work earlier in this project, and his help on implementing parts of the garbage collector presented here. Our work is supported by the National Science Foundation under Grants CCF–1544542, CCF–1318227, CCF–1618732, ONR award 503353, the Agence Nationale de la Recherche (ANR) under Grant 14-CE28-0004, and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement 695412).

References

1. Brookes, S.: A semantics for concurrent separation logic. Theoretical Computer Science **375**(1-3) (2007)
2. Davis, J., Myreen, M.O.: The reflective milawa theorem prover is sound (down to the machine code that runs it). Journal of Automated Reasoning **55**(2) (2015)
3. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the Theory and Practice of Software (TACAS) (2008)
4. Demange, D., Laporte, V., Zhao, L., Jagannathan, S., Pichardie, D., Vitek, J.: Plan B: a buffered memory model for Java. In: Symposium on Principles of Programming Languages (POPL) (2013)
5. Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S., Steffens, E.F.M.: On-the-fly garbage collection: An exercise in cooperation. Communications of the ACM **21**(11) (1978)
6. Doligez, D., Gonthier, G.: Portable, unobtrusive garbage collection for multiprocessor systems. In: Symposium on Principles of Programming Languages (POPL) (1994)
7. Doligez, D., Leroy, X.: A concurrent, generational garbage collector for a multithreaded implementation of ML. In: Symposium on Principles of Programming Languages (POPL) (1993)
8. Domani, T., Kolodner, E.K., Lewis, E., Salant, E.E., Barabash, K., Lahan, I., Levanoni, Y., Petrank, E., Yanover, I.: Implementing an on-the-fly garbage collector for Java. In: International Symposium on Memory Management (ISMM) (2000)
9. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: Symposium on Principles of Programming Languages (POPL) (2009)
10. Gammie, P., Hosking, A.L., Engelhardt, K.: Relaxing safely: verified on-the-fly garbage collection for x86-TSO. In: Programming Language Design and Implementation (PLDI) (2015)
11. Gonthier, G.: Verifying the safety of a practical concurrent garbage collector. In: International Conference on Computer Aided Verification (CAV) (1996)
12. Havelund, K.: Mechanical verification of a garbage collector. In: International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP) (1999)
13. Hawblitzel, C., Howell, J., Lorch, J.R., Narayan, A., Parno, B., Zhang, D., Zill, B.: Ironclad apps: End-to-end security via automated full-system verification. In: Conference on Operating Systems Design and Implementation (OSDI) (2014)
14. Hawblitzel, C., Petrank, E.: Automated verification of practical garbage collectors. In: Symposium on Principles of Programming Languages (POPL) (2009)
15. Hawblitzel, C., Petrank, E., Qadeer, S., Tasiran, S.: Automated and modular refinement reasoning for concurrent programs. In: International Conference on Computer Aided Verification (CAV) (2015)
16. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. Transactions on Programming Languages and Systems (TOPLAS) **12**(3) (1990)
17. Jagannathan, S., Laporte, V., Petri, G., Pichardie, D., Vitek, J.: Atomicity refinement for verified compilation. Transactions on Programming Languages and Systems (TOPLAS) **36**(2) (2014)
18. Jones, C.B.: Tentative steps toward a development method for interfering programs. Transactions on Programming Languages and Systems (TOPLAS) **5**(4) (1983)
19. Jones, R., Hosking, A., Moss, E.: The Garbage Collection Handbook. Chapman & Hall (2011)
20. Leino, K.R.M.: This is boogie 2 (2008)

21. Liang, H., Feng, X., Fu, M.: A Rely-Guarantee-based simulation for verifying concurrent program transformations. In: Symposium on Principles of Programming Languages (POPL) (2012)
22. Liang, H., Feng, X., Fu, M.: Rely-Guarantee-based simulation for compositional verification of concurrent program transformations. Transactions on Programming Languages and Systems (TOPLAS) **36**(1) (2014)
23. McCreight, A., Chevalier, T., Tolmach, A.: A certified framework for compiling and executing garbage-collected languages. In: International Conference on Functional Programming (ICFP) (2010)
24. McCreight, A., Shao, Z., Lin, C., Li, L.: A general framework for certifying garbage collectors and their mutators. In: Programming Language Design and Implementation (PLDI) (2007)
25. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Symposium on Principles of Distributed Computing (PODC) (1996)
26. Myreen, M.O.: Reusable verification of a copying collector. In: Conference on Verified Software: Theories, Tools, Experiments (VSTTE) (2010)
27. O'Hearn, P.W.: Separation logic and concurrent resource management. In: International Symposium on Memory Management (ISMM) (2007)
28. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs i. Acta Inf. **6**(4) (1976)
29. Pavlovic, D., Pepper, P., Smith, D.R.: Formal derivation of concurrent garbage collectors. In: International Conference on Mathematics of Program Construction (MPC) (2010)
30. Pfenning, F., Elliott, C.: Higher-order abstract syntax. In: Programming Language Design and Implementation (PLDI) (1988)
31. Pizlo, F., Ziarek, L., Maj, P., Hosking, A.L., Blanton, E., Vitek, J.: Schism: fragmentation-tolerant real-time garbage collection. In: Programming Language Design and Implementation (PLDI) (2010)
32. Prensa Nieto, L.: The Rely-Guarantee method in Isabelle/HOL. In: European Conference on Programming (ESOP) (2003)
33. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Symposium on Logic in Computer Science (LICS) (2002)
34. Sandberg Ericsson, A., Myreen, M.O., Aman Pohjola, J.: A Verified Generational Garbage Collector for CakeML (2017)
35. Sergey, I., Nanevski, A., Banerjee, A.: Mechanized verification of fine-grained concurrent programs. In: Programming Language Design and Implementation (PLDI) (2015)
36. Ševčík, J., Vafeiadis, V., Nardelli, F.Z., Jagannathan, S., Sewell, P.: CompcertTSO: A verified compiler for relaxed-memory concurrency. Journal of the ACM **60**(3) (2013)
37. Torp-Smith, N., Birkedal, L., Reynolds, J.C.: Local reasoning about a copying garbage collector. Transactions on Programming Languages an Systems (TOPLAS) **30**(4) (2008)
38. Treiber, R.K.: Systems programming: Coping with parallelism. Tech. rep., IBM Almaden Research Center (1986)
39. Vafeiadis, V.: Concurrent separation logic and operational semantics. Electronic Notes in Theoretical Computer Science **276** (2011)
40. Vafeiadis, V., Parkinson, M.J.: A marriage of rely/guarantee and separation logic. In: International Conference on Concurrency Theory (CONCUR) (2007)
41. Yang, J., Hawblitzel, C.: Safe to the last instruction: Automated verification of a type-safe operating system. In: Programming Language Design and Implementation (PLDI) (2010)
42. Zakowski, Y., Cachera, D., Demange, D., Petri, G., Pichardie, D., Jagannathan, S., Vitek, J.: Verifying a concurrent garbage collector using a Rely-Guarantee methodology – companion website (2017). http://www.irisa.fr/celtique/ext/cgc/
43. Zakowski, Y., Cachera, D., Demange, D., Pichardie, D.: Compilation of linearizable data structures - a mechanised RG logic for semantic refinement. In: Symposium on Applied Computing (SAC) (2018)